# Fix-point Equations for Well-Founded Recursion in Type Theory

Antonia Balaa and Yves Bertot

INRIA Sophia-Antipolis
http://www-sop.inria.fr/lemme/{Antonia.Balaa,Yves.Bertot}

## 1 Introduction

Inductive type theories, as used in systems like Coq or Lego [11, 14, 4], provide a systematic approach to program recursive functions over inductive data-structures and to reason about these functions. Recursive computation is described by reduction rules, included in the type system under the name $\iota$-*reduction*. If $t$ is an element of a recursive type, $f$ is a recursive function over that type and $v$ is the value of $f(t)$, then the equality $f(t) = v$ is a simple tautology, because $f(t)$ and $v$ are equal modulo $\iota$-reduction.

In practice, the inductive data-structures on which recursive functions may be defined are not restricted to simple concrete data-types, such as term algebras or finite-branching trees. Infinite branching is also allowed, so that some inductive types are powerful enough to describe the more complicated notions of terminating functions [9]. The intuitive notion is that of well-founded orders, for which there exists no infinite descending chain. If we describe a function such that recursive calls are performed only on terms that are smaller than the initial argument for some well-founded order, then we are sure there cannot be an infinite sequence of recursive calls.

For a user intending to describe complex programs, this is good news: it becomes possible to write algorithms without having to follow a tedious encoding using primitive structural recursion. Pleasant and efficient descriptions of complex algorithms become possible, for example, algorithms for computing Gröbner bases, as described by Théry [16] and Coquand and Persson [5].

In practice, using a function defined by well-founded recursion can also be unwieldy. In some sense, these functions are also defined by structural recursion, but the recursion follows the structure of the proof that there is no infinite descending chain starting from the function's argument. To actually use the reduction rules, one needs a full description of how the proof was constructed and it is not enough to simply know that such a proof exists.

This work provides a fix-point equality theorem that represents the reduction rule and can be used without any knowledge of the proofs' structure. The first motivation is practical in nature. Without a fix-point equality, it is hard to collect information about a function. In a system like Coq, the fix-point equation is only a consequence of the encoding based on generalized inductive data-types. With our work to generate this equality automatically, it becomes simpler to reason about well-founded recursive functions in Coq.

### 1.1 Related work

This work draws its theoretical and practical background from all the work done around inductive definitions in mathematics and computer based theorem proving tools. From the theoretical point of view, the main entry point may be the work by P. Aczel [1]. The immersion of inductive definitions in type theory based systems was mostly studied in the 1980 decade, for instance by Constable and Mendler in [3, 2]. The flavor we use in this paper is mostly described by Coquand, Pfenning, and Paulin-Mohring in [14, 4, 11]. General use of well-founded recursion in Martin-Löf's intuitionistic type theory was studied by Paulson in [12], who shows that reduction rules can be obtained for each of several means to construct well-founded relations from previously known well-founded relations. By comparison with Paulson's work, our technique is to obtain reduction rules that are specific to each recursive function. The introduction of well-founded recursion using an accessibility principle as used in this paper was described by Nordström in [9].

Inductive definitions and inductive types also appear in proof systems based on simply-typed higher-order logic, such as HOL [6] or Isabelle [13]. Camilleri and Melham provide a package to systematize the definition of inductive relations in the HOL system [8], but this is not powerful enough to describe the notion of accessibility used in this paper. Harrison [7] provides a more practical tool, powerful enough to describe the necessary elements for well-founded recursion as described in this paper. In particular, the transfer theorem described below in section 3.1 is also described in Harrison's work. Konrad Slind [15] also gives a package to ease the definition of recursive functions based on well-founded induction, making sure his package will be general enough to be used with two different proof systems. The work of Harrison and Slind is also gives solutions to the question of constructing fix-point equations (they are called *recursion equations* in [15]), but the main question here is whether the logical foundations of type-theory, with constructivity and without extensionality, is sufficient to recover these equations. Harrison and Slind do not cover this question, using extensionality and non-constructive features freely.

## 2 Detailed description of the problem

In this section, we describe the inductive objects of type theory as used in $CC^{ind}$ [11]. We show the differences that exist between structural recursive functions and well-founded recursive functions.

In type theory, types are used to represent both propositions and datatypes. To distinguish between these, *sorts* are provided, that are used to denote the types of types. The sort *Set* will be used to type the types representing datatypes, while the sort *Prop* will be used for the types representing propositions. Inductive definitions can be used to construct inductive datatypes, for instance the type of natural numbers given below, or to construct inductive propositions, for instance the proposition of accessibility given below. Types can be returned by functions. When the returned value is in *Prop*, a function is actually a proposition, will shall used the letter $P$ to range over such propositions. When the returned value is in *Set*, a function denotes a dependent type. We shall use the letter $B$ to range over such functions.

An inductive type is specified by giving a name, a type, and the name and type of its constructors. For instance, the *nat* for natural numbers is constructed using an inductive definition with two constructors 0 and $S$.

$$\begin{aligned} &\mathit{Inductive}\ \mathit{nat}\ : \mathit{Set} := \\ &\qquad 0 : \mathit{nat} \\ &\qquad |S : \mathit{nat} \rightarrow \mathit{nat}. \end{aligned}$$

It is possible to define structurally recursive functions over an inductive type. For a given function, the user needs to provide values for the constructors that are not really recursive and functions that compute the final value from the values of recursive calls of sub-terms belonging to the inductive type for the other constructors. For natural numbers, there exists an operation *nat_rec* whose behavior is described by the following two reduction rules (we use the notation $(f\ x_1\ x_2)$ to represent the application of $f$ to two arguments):

$$(\mathit{nat\_rec}\ t\ V_0\ f\ 0) \rightsquigarrow V_0$$

$$(\mathit{nat\_rec}\ t\ V_0\ f\ (S\ x)) \rightsquigarrow (f\ x\ (\mathit{nat\_rec}\ t\ V_0\ f\ x))$$

The term *nat_rec* is a plain object of the typed calculus, its type takes into account the fact that one may define a function with a dependent type $B$. The $\Pi$ notation is used to express that the type of the value returned by a function depends on the value received as argument by this function. When the type is used as a proposition, we will replace the $\Pi$ notation with a more intuitive $\forall$.

$$\begin{aligned} &\mathit{nat\_rec} : \\ &\Pi\ B : \mathit{nat} \rightarrow \mathit{Set}. \\ &\quad (B\ 0) \rightarrow \\ &\quad (\Pi x :\ \mathit{nat}\ (B\ x) \rightarrow (B\ (S\ x))) \rightarrow (\Pi\ x :\ \mathit{nat}.B(x)). \end{aligned}$$

Inductive types may be more general. Consider the notion of *accessibility*, as described in [9], and defined as follows: given a binary relation $R$ over a set $A$, an element $a \in A$ is accessible if every element smaller than $a$ (we use a terminology where $R$ is used as an order, but it does not even need to be transitive) is accessible. In particular, all minimal elements are accessible. The notion of accessibility can be defined with the following inductive definition:

$$\begin{aligned} &\mathit{Inductive}\ \mathit{Acc}[A : \mathit{Set};\ R : A \rightarrow A \rightarrow \mathit{Prop}] : A \rightarrow \mathit{Prop} := \\ &\mathit{Acc\_intro} : \forall x : A.(\forall y : A.(R\ y\ x) \Rightarrow (\mathit{Acc}\ A\ R\ y)) \Rightarrow (\mathit{Acc}\ A\ R\ x). \end{aligned}$$

To make the notation easier, we shall assume that we work with a fixed relation, which we shall denote $<$ instead of $R$. Here, the recursion operation is described by a single reduction rule:

$$\begin{aligned} &(\mathit{Acc\_rec}\ t\ \Phi\ x\ (\mathit{Acc\_intro}\ x\ h)) \rightsquigarrow \\ &\qquad\qquad (\Phi\ x\ h\ \lambda y : A.\lambda p : y < x.(\mathit{Acc\_rec}\ t\ \Phi\ y\ (h\ y\ p))) \end{aligned}$$

*Acc_rec* takes the following type:

$$\begin{array}{l} Acc\_rec : \Pi B : A \rightarrow Set. \\ \quad (\Pi x : A. \\ \qquad (\forall y : A.y < x \Rightarrow (Acc\ y)) \rightarrow \\ \qquad (\Pi y : A.y < x \rightarrow (B\ y)) \rightarrow (B\ x)) \rightarrow \\ \quad \Pi x : A.(Acc\ x) \rightarrow (B\ x) \end{array}$$

We often omit the first argument of *Acc_rec*, represented here by the formal parameter $B$ of type: $A \rightarrow Set$. The function $\Phi$ must have the following type:

$$\begin{array}{l} \Phi : (\Pi x : A.(\forall y : A.y < x \Rightarrow (Acc\ y)) \rightarrow \\ \qquad (\Pi y : A.y < x \rightarrow (B\ y)) \rightarrow (B\ x)) \end{array}$$

The function $\Phi$ takes three values as arguments, an object $x$, a proof that all elements smaller than $x$ are accessible, and a function that can only be called on objects that are smaller than $x$. The reduction rule expresses that when computing the value $(Acc\_rec\ t\ \Phi\ x\ q)$, you can unroll the recursion one step, replacing it with a call to $\Phi$ on the right arguments, but you can perform this step only when $q$ is of the form $(Acc\_intro\ x\ h)$.

Accessibility describes those elements from which one cannot start an infinite descending chain. A relation is well-founded when the accessibility predicate is verified for all elements of the type. There is then a theorem $wf_<$ that states $wf_< : \forall x : A.(Acc\ x)$. With a well-founded relation, it is possible to construct a new function $Rec_{wf<}$ with the following value:

$$\begin{array}{l} (Rec_{wf<}\ F\ x) = \\ \quad (Acc\_rec\ (\lambda z : A.\lambda h : (\forall y : A.y < z \Rightarrow (Acc\ z)).(F\ z))\ x\ (wf_<\ x)) \end{array}$$

The function $F$ must have the following type:
$F : \Pi x : A.(\Pi y : A.y < x \rightarrow (B\ y)) \rightarrow (B\ x)$.

This function $Rec_{wf<}$ is used to define functions by well-founded recursion. But does it have a reduction behavior that is similar to a recursion operator? At first sight no, since the proof that $x$ is accessible is of the form $(wf_<\ x)$, not an instance of *Acc_intro*, the form that would be needed to use the $\iota$-reduction of *Acc_rec*. Under closer scrutiny however, it is possible to prove that every object of an inductive type is obtained by applying one of the constructors of this type to proper arguments. Here there is only one constructor to the type *Acc* and one can deduce there exists a hypothesis $h$ stating $\forall y : A.y < x \Rightarrow (Acc\ x)$ such that the proof $(wf_<\ x)$ is equal to $(Acc\_intro\ x\ h)$. Based on this equality, we have the following rule (with some type information omitted):

$$(Rec_{wf<}\ F\ x) \leadsto (F\ x\ \lambda y : A.\lambda p : y < x.(Acc\_rec(\lambda x \lambda p'.(F\ x))\ y\ (h\ y\ p'))) \quad (1)$$

What is unfortunate with this rule is the presence of *Acc_rec* instead of $Rec_{wf<}$ in the right hand side. Certainly, a rule of the following form would be preferable:

$$(Rec_{wf<}\ F\ x) \leadsto (F\ x\ \lambda y : A.\lambda p : y < x.(Rec_{wf<}\ F\ y)) \quad (2)$$

It is the objective of this work to provide a systematic method to provide rule 2 as an equality theorem.

## 3 Producing the fix-point equation

We can break down the problem into two parts. The first part shows that proving the fix-point equation can be reduced to proving a specific property of the function $F$. The second part describes a method to prove that property for a class of functions.

### 3.1 Reducing to the step hypothesis

The crux of fix-point equation proofs is to show that the values of proofs really are irrelevant. For any value $x$ in $A$, there exists a term $T$ obtained by a certain number $n_x$ of unrollings of $F$, such that $(Acc\_rec\ \lambda y.\lambda h.(F\ y)\ x\ h) = (T\ x)$. Thus, if $F$ does not use the actual value of proof arguments, $(Acc\_rec\ \lambda y.\lambda p.(F\ y)\ x\ h)$ does not either. Since the term $h$ in this expression is a proof, its value is not used and it does not matter whether $h$ is of the form $(wf_<\ x)$ or $(h'\ x\ h'')$.

The transfer theorem has the following statement:

$$\begin{array}{l}
\forall F : \Pi x : A.(\Pi y : A.y < x \to (B\ y)) \to (B\ x).\\
\quad (\forall x : A.\forall f : (\Pi y : A.(B\ y)).\forall g : (\Pi y : A.y < x \to (B\ y)).\\
\qquad (\forall y : A.\forall h : y < x.(g\ y\ h) = (f\ y))\\
\qquad \Rightarrow (F\ x\ \lambda y : A.\lambda h : (y < x).(g\ y\ h)) =\\
\qquad\quad (F\ x\ \lambda y : A.\lambda h : (y < x).(f\ y)))\\
\Rightarrow \forall x : A.(Rec_{wf<}\ F\ x) = (F\ x\ \lambda y : A \lambda p : y < x.(Rec_{wf<}\ F\ y)).
\end{array}$$

The proof of this theorem proceeds by an induction over the accessibility of $x$ with respect to the transitive closure of $<$. See appendix A for a Coq proof.

We shall call *step hypothesis* the only hypothesis of this transfer theorem.

### 3.2 Using extensionality

The axiom of extensionality states that two functions are equal as soon as their values are equal on every possible argument. It has the following statement:

$$\forall A : Type.\forall B : A \to Set.\forall f, g : (\Pi x : A.(B\ x)).(\forall x : A.(f\ x) = (g\ x)) \Rightarrow f = g$$

Obviously, the step hypothesis is a direct consequence of extensionality. If this axiom is added, then the question of the fix-point equation is solved. The rest of this paper describes how one can avoid using extensionality.

## 4 A syntactic construction of the step hypothesis

For a given $F$, the step hypothesis has the following statement:

$$\begin{array}{l}
(\forall x : A.\forall f : (\Pi y : A.(B\ y)).\forall g : (\Pi y : A.y < x \to (B\ y)).\\
\qquad (\forall y : A.\forall h : y < x.(g\ y\ h) = (f\ y))
\end{array}$$

$$\Rightarrow (F\ x\ \lambda y : A.\lambda h : (y < x)(g\ y\ h)) =$$
$$(F\ x\ \lambda y : A.\lambda h : (y < x).(f\ y)))$$

Constructing a proof of this statement without using extensionality will be done by a structural analysis of $F$.

### 4.1 Some notation

Given a set of sorts $\mathcal{S}$ ($\mathcal{S}$ will typically contain categories like $Prop$, $Set$, $Type$), a *term* is an element of the language $\mathcal{T}$ defined as follows:

$$\mathcal{T} ::= \lambda x : \mathcal{T}.\mathcal{T} | (\mathcal{T}\mathcal{T}) | \Pi x : \mathcal{T}.\mathcal{T} | \mathcal{S} | x.$$

This is the usual set of terms for type theory, see for example in section 2 in [11].

We will not describe the typing rules for this language here, but we shall assume we work in a context where a collection of inductive types has been defined and all the terms we manipulate are well-typed. In particular, for every inductive type $ty$ we will assume there exists a function corresponding to a case operator, and a function corresponding to reasoning by cases, whose types are obtained systematically from the inductive definition following the methods in [11]. The case operator is a simplified instance of the recursion operator, where recursive values are not used, while the "reasoning-by-cases" operator is deduced similarly from the induction principle associated to the inductive type.

For instance, in the case of natural numbers, the case operator is the function $nat\_case$ with the following type:

$nat\_case$ :
$$\Pi B : nat \to Set.(B\ 0) \to (\Pi m : nat.(B\ (S\ m))) \to \Pi n : nat.(B\ n)$$

The "reasoning-by-case" operation has the following type:

$nat\_case'$ :
$$\forall P : nat \to Prop.(P\ 0) \to (\forall m : nat.(P\ (S\ m))) \to \forall n : nat.(P\ n)$$

For an arbitrary type $ty$, we will suppose it has $k$ constructors

$$c_i : \Pi x_1 : t_{i,1} \ldots \Pi x_{n_i} : t_{i,n_i}.ty \qquad (1 \leq i \leq k).$$

The case operator for this inductive type will be denoted $ty\_case$, with the following type:

$ty\_case : \Pi B : ty \to Set.$
$$(\Pi x_1 : t_{1,1} \ldots \Pi x_{n_1} : t_{1,n_1}.(B\ (c_1\ x_1 \ldots x_{n_1}))) \Rightarrow$$
$\ldots$
$$(\Pi x_1 : t_{k,1} \ldots \Pi x_{n_k} : t_{k,n_k}.(B\ (c_k\ x_1 \ldots x_{n_k}))) \Rightarrow$$
$\Pi x : ty.(B\ x)$

When considering a usage of $ty\_case$, it will have the form

$$(ty\_case\ t\ l_1 \ldots l_k\ V)$$

Where each $l_i$ will have the form:

$$l_i = \lambda x_1 : t_{i,1} \ldots \lambda x_{n_i} : t_{i,n_i}.b_i$$

and we shall sometimes abbreviate this expression as:

$$l_i = \lambda\overline{x_{(i)}}.b_i$$

For the type *ty*, we shall also assume there exists a term *ty_case'* with type

$$\begin{array}{l} ty\_case' : \Pi P : ty \to Prop. \\ \qquad (\Pi x_1 : t_{1,1} \ldots \Pi x_{n_1} : t_{1,n_1}.(P\ (c_1\ x_1 \ldots x_{n_1}))) \Rightarrow \\ \ldots \\ \qquad (\Pi x_1 : t_{k,1} \ldots \Pi x_{n_k} : t_{k,n_k}.(P\ (c_k\ x_1 \ldots x_{n_k}))) \Rightarrow \\ \Pi x : ty.(P\ x). \end{array}$$

This term *ty_case'* will be used to construct proofs instead of values.

We will also assume there exists two theorems *eq_ind_r* and *eq_ind*, with the following statements (types):

$$eq\_ind\_r : \forall A : Set.\forall x : A.\forall P : A \to Prop.(P\ x) \Rightarrow \forall y : A.y = x \Rightarrow (Py),$$

$$eq\_ind : \forall A : Set.\forall x : A.\forall P : A \to Prop.(P\ x) \Rightarrow \forall y : A.x = y \Rightarrow (Py),$$

and a theorem $refl\_equal$ with the following statement:

$$refl\_equal : \forall A : Set.\forall x : A.x = x.$$

On several occasions, we will consider *contexts* which are expressions with a hole in them, to be filled with some value. A context will be denoted $C[\cdot; \ldots; \cdot]$ and the same contexts where holes have been filled in with values will be denoted $C[x_1; \ldots; x_n]$. We will also consider the operation of substituting all occurrences of a term with another. This operation will be denoted $C[g := h]$. This term is equal to $C[h; \cdots; h]$, where $C[\cdot; \ldots; \cdot]$ is the context such that $C[g; \ldots; g] = C$ and $g$ does not occur free in $C[\cdot; \ldots; \cdot]$.

### 4.2 Constructing the proof in absence of bound variables

For some values of $X$, computing $(F\ X\ g)$ may lead to some term where $g$ occurs. Let us say it is convertible to

$$T = C[(g\ T'\ H)],$$

such that all variables occurring in the terms $T'$ and $H$ are free in $T$. Suppose we have an hypothesis $H_{eq}$

$$H_{eq} : \forall y : A.\forall h : y < x.(g\ y\ h) = (f\ y).$$

Let $C' = C[g := \lambda y : A.\lambda h : y < x.(fy)]$, we can reduce the task of proving $C[(g\ T'\ H)] = C'$ to the task of proving $C[(f\ T')] = C'$ using $eq\_ind\_r$ and $H_{eq}$. If $p$ is a proof of $C[(f\ T')] = C'$ , then

$$(eq\_ind\_r\ (B\ T')\ (f\ T')\ (\lambda n.C[n] = C')\ p\ (g\ T'\ H)\ (H_{eq}\ T'\ H))$$

is a proof of $C[(g\ T'\ H)] = C'$. We can repeat this process until there is no more instance of $g$ in the left-hand side, and conclude using $refl\_equal$. Note the constraint about free variables in $T'$ and $H$ that must also be free in $C[(g\ T'\ H)]$. If this constraint is not respected the proof term will not be well typed.

### 4.3 Handling case operators

Bound variables are introduced when one uses the case operator of some inductive type. Intuitively, the case operators perform some form of pattern matching on values in an inductive type. This pattern matching makes several cases appear, corresponding to the various constructors of the inductive type. Most constructors carry values and the $\lambda$-abstractions are used in the case operators to give access to these values, like pattern-matching rules in functional programming.

For instance, let us consider a well-founded definition of a discrete logarithm function:

$$\begin{array}{l}
(F_{log}\ x) = \\
\quad (nat_case\ \lambda x : nat.\Pi f : (\Pi y : nat.(y < x) \to nat).nat \\
\qquad\qquad \lambda f : (\Pi y : nat.(y < 0) \to nat).0 \\
\qquad\qquad \lambda\, n : nat.\lambda f : (\Pi y : nat.(y < (S\ n)) \to nat).(S\ (f\ (div2\ (S\ n))\ (th\ n))) \\
log = (Rec_{wf<}\ F_{log})
\end{array}$$

This definition assumes we are given a function $div2$ that divides a number by 2 and a theorem $th : \forall n : nat.(div2\ n) < (S\ n)$. The second and third arguments of $nat\_case$ describe the computation to perform in two different cases. This is where $\lambda$-abstractions occur. The bound variable $n$ represents the predecessor of the function argument in the case where this argument is non-zero.

The idea for this part of our work is to follow the same structure of pattern matching, not to compute the value returned by the function, but to construct a different proof of equality in each case. The recursive analysis of $F$ is decomposed into two phases. The first phase corresponds to the subterms of $F$ that have a functional type, receiving a function as argument. The second part is the part where the function argument is already received. One switches from the first phase to the second when crossing a $\lambda$-abstraction.

**First phase : sub-terms of $F$ with a functional argument.** The expression $F$ has type $\Pi x : A.(\Pi y : A.y < x \to (B\ y)) \to (B\ x)$. It is a function that can receive (at least) two arguments. We shall always assume that $F$ is of the form $\lambda x.F'$.

The expression $F'$ must be well typed in a context where $x$ has type $A$ and $F'$ must have the type:

$$\Pi y : A.y < x \rightarrow (B\ y)) \rightarrow (B\ x).$$

In a more general setting, the problem is the following one: given a type environment $\Gamma$, a term $X$ well-typed in $\Gamma$, and composed only of applications of constructors to other constructors and variables occurring in $\Gamma$, given a variable $f$ declared in $\Gamma$ with type $f : \Pi y : A.(B\ y)$, prove the following equality:

$$\begin{aligned}&\forall g : (\Pi y : A.y < X \rightarrow (B\ y)).(\forall y : A.\forall h : y < X.(g\ y\ h) = (f\ y)) \Rightarrow\\ &\qquad (F'\ \lambda x : A.\lambda h : y < X.(f\ y)) = (F'\ \lambda x : A.\lambda h : y < X.(g\ y\ h)\end{aligned}$$

$F'$ should be a function, but it would be too restrictive to assume that $F'$ should be a $\lambda$-abstraction. In fact we shall consider two cases:

1. it is a $\lambda$-abstraction, $F' = \lambda g.T$, in this case we can switch to the second phase for $T$, after having added in the context $g : \Pi y : A.y < X \rightarrow (B\ y)$ and $H_{eq} : \forall y : A.\forall h : y < X.(g\ y\ h) = (f\ y)$. If this yields a proof $p$, then

$$\lambda g : \Pi y : A.y < X \rightarrow (B\ y).\lambda H_{eq} : \forall y : A.\forall h : y < X.(g\ y\ h) = (f\ y).p$$

   is the requested proof.
2. it is built with a *case* expression, that is, $F'$ has the following form:

$$F' = (ty\_case\ t\ l_1\ \ldots l_k\ V).$$

   where $t$, $l_1$, $\ldots$, $l_k$ are formed as described in section 4.1. Because of the type of $F'$, the term $t$ should have the following form:

$$t = \lambda x : ty.\Pi g : (\Pi y : A.y < X[x] \rightarrow (B\ y)).t'$$

   where $X[\cdot]$ is a context such that $X[V] = X$ and $t'$ is some arbitrary type. Note that $X[\cdot]$ may have no hole when $V$ does not occur in $X$.
   Recall the terms $l_i$ are such that $l_i = \lambda\overline{x_{(i)}}.b_i$, we can recursively address the problem of finding a proof for formulas:

$$\begin{aligned}&\forall g : (\Pi y : A.\Pi h : y < X[(c_i\ x_1 \ldots x_{n_1})].(B\ y)).\\ &\qquad (\forall y : A.\forall h : y < X[(c_i\ x_1 \ldots x_{n_i})].(g\ y\ h) = (f\ y)) \Rightarrow\\ &\qquad\qquad (b_i\ \lambda y : A.\lambda h : y < X[(c_i\ x_1 \ldots x_{n_i})](f\ y)) = (b_i\ g)\end{aligned}$$

   now with the term $X$ being replaced with $X[(c_i\ x_1 \ldots x_{n_i})]$, a new type environment $\Gamma'$ modified so that the new variables $x_1$, $\ldots$, $x_n$ have been added with type $x_j : t_{i,j}$. Let us suppose this yields $k$ proofs called $p_i$. We can then construct a proof for the whole equality with the following shape:

$$\begin{aligned}&(ty\_case'\\ &\qquad \lambda x : ty.\forall g : (\Pi y : A.y < X[x] \rightarrow (B\ y)).\\ &\qquad\qquad (\forall y : A.\forall h : y < X[x].(g\ y\ h) = (F\ y)) \Rightarrow\\ &\qquad\qquad (ty\_case\ t\ l_1 \ldots l_k\ x\ \lambda y : A.\lambda h : y < X[x].(g\ y\ h)) =\end{aligned}$$

$$\begin{array}{l} \quad\quad\quad (ty\_case\ t\ l_1 \ldots l_k\ x\ \lambda y : A.\lambda h : y < X[x].(f\ y)) \\ \lambda \overline{x_{(1)}}.p_1 \\ \ldots \\ \ldots \\ \lambda \overline{x_{(k)}}.p_k \\ V\ g\ H_{eq}) \end{array}$$

**Second phase: sub-terms of F where $g$ occurs.** In the second phase, the variable $g$, corresponding to the function that may be called to represent recursive calls, has a special status, and we are looking at a sub-term of $F$ where $g$ may occur free. We want to provide the equality of this term with another term where all free occurrences of $g$ are replaced by $\lambda y : A.\lambda h.y < X.(f\ y)$.

We reduce the problem to the following one: given a type environment $\Gamma$, a term $X$ well-typed in $\Gamma$, and composed only of application of constructors to other constructors and variables occurring in $\Gamma$, given three variables $f$, $g$, and $H_{eq}$ declared in $\Gamma$ with types $f : \Pi y : A.(B\ y)$, $g : \Pi y : A.\Pi h : y < X.(B\ y)$, and $H_{eq} : \forall y : A.\forall h : y < X.(g\ y\ h) = (f\ y)$, given an expression $C$, well typed in $\Gamma$ where $f$ does not occur, construct a proof for the equality:

$$C = C[g := \lambda y.\lambda h : y < X.(f\ y)].$$

We only consider the following cases:

1. if $C$ falls in the case described in section 4.2, then apply the corresponding method,
2. if $C = (g\ T\ H)$, compute recursively a proof $p$ for

$$T = T[g := \lambda y.\lambda h : y < X.(f\ y)],$$

   then the following expression is a good candidate for the requested proof:

$$\begin{array}{l} (eq\_ind\ A\ T\ [n : A](g\ T\ H) = (f\ n)\ (H_{eq}\ T\ H) \\ \quad\quad T[g := \lambda y.\lambda h : y < X.(f\ y)]\ p) \end{array}$$

   However, this expression may be untypable. The type of this expression is supposed to be $(g\ T\ H) = (f\ T')$, but this expression is well-typed if its type is well-typed. For an equality to be well-typed, both sides must have the same type. The type of $(g\ T\ H)$ is $(B\ T)$ and the type of $(f\ T')$ is $(B\ T')$. If these two expressions are not $\beta\iota$-convertible, our method will fail.
3. if $C = (C'\ g)$ and $g$ does not occur in $C'$, then switch back to the first phase to find a proof of:

$$\begin{array}{l} \forall g : (\Pi y : A.(y < X).(B\ y)).(\forall y : A.\forall h : (y < X).(g\ y\ h) = (f\ y)) \Rightarrow \\ (C'\ g) = (C'\ \lambda y : A.\lambda h : y < X.(f\ y)), \end{array}$$

   then apply this proof to the current $g$ and $H_{eq}$.

4. if $C = (ty\_case\ t\ l_1 \ldots l_k\ V)$ and $ty, l_1, \ldots, l_k$ are described as in section 4.1, let us suppose we are recursively able to find a proof $p_0$ of

$$V = V[g := \lambda y.\lambda h : y < X.(f\ y)]$$

in the same context $\Gamma$, for the same value $X$, and the same $f$, $g$, and $H_{eq}$, and let us suppose we are recursively able to find $k$ proofs $p_i$ for the following statements:

$$b_i = b_i[g := \lambda y.\lambda h : y < X'_i.(f\ y)]$$

where $X'_i$ is $X[V := (c_i\ x_1 \ldots x_k)]$, with new typing contexts $\Gamma_i$ such that declarations have been added for each of the variables $x_1 : t_{i,1}, \ldots, x_{n_i} : t_{i,n_i}$ (modulo a renaming of variables to make sure the new variables are distinct from variables already present in $\Gamma$), and where the two variables $g$, $H_{eq}$ have their type changed in $\Gamma_i$ to

$$g : \lambda y : A.y < X'_i \to (B\ y) \qquad H_{eq} : \lambda y : A.\lambda h : y < X'_i.(g\ y\ h) = (f\ y).$$

Writing $l'_i$ for the terms $\lambda\overline{x_{(i)}}.b_i[g := \lambda y.\lambda h : y < X'_i.(f\ y)]$, the term

$$\begin{array}{l}
p = (ty_case' \\
\quad \lambda x : ty.\forall g : (\Pi y : A.y < X[V := x] \to (B\ y)). \\
\qquad \forall H_{eq} : \forall y : A.\forall h : y < X[V := x].(g\ y\ h) = (f\ y). \\
\qquad\quad (ty_case\ t\ l_1 \ldots l_k\ x) = (ty_case\ t\ l'_1 \ldots l'_k\ x) \\
\quad \lambda x_1 : t_{1,1}. \ldots \lambda x_{n_1} : t_{1,n_1}.\lambda g : (\Pi y : A.y < X'_1 \to (B\ y)). \\
\qquad \lambda H_{eq} : (\forall y : A.\forall h : y < X'_1 \Rightarrow (f\ y) = (g\ y\ h)).p_1 \\
\quad \ldots \\
\quad \lambda x_1 : t_{k,1}. \ldots \lambda x_{n_k} : t_{k,n_k}.\lambda g : (\Pi y : A.y < X'_k \to (B\ y)). \\
\qquad \lambda H_{eq} : (\forall y : A.\forall h : y < X'_k \Rightarrow (f\ y) = (g\ y\ h)).p_k \\
\quad V\ g\ H_{eq})
\end{array}$$

is a proof of:

$$(ty\_case\ t\ l_1 \ldots l_k\ V) = (ty\_case\ l'_1 \ldots l'_k\ V)$$

and

$$\begin{array}{l}
(eq_ind_r\ ty \\
\quad V \\
\quad \lambda x : ty.(ty_case\ t\ l_1 \ldots l_k\ V) = (ty_case\ l'_1 \ldots l'_k\ x) \\
\quad p \\
\quad V[g := \lambda y.\lambda h : y < X.(f\ y)] \\
\quad p_0)
\end{array}$$

is a proof of $C = C[g := \lambda y.\lambda h : y < X.(f\ y)]$.

### 4.4 The class of acceptable functions

We put several restrictions on the form of functions for which our method will work. These restrictions can be summarized as follows: the function $F$ must be well-typed and must only use its functional argument in two possible ways:

1. The functional argument is fully applied
2. The functional argument is itself passed as argument to a case construct, that is, a function of the form *ty_case*.

In particular, this precludes situations where $F$ would pass its functional argument to an auxiliary function. Attempting to summarize these constraints as a formal language description we have the following result:

$$\begin{aligned}
\mathcal{T}' &= \lambda x : \mathcal{T}.\mathcal{T}_1\\
\mathcal{T}_1 &= \lambda f : (\Pi y : \mathcal{T}.\Pi h : (R\ y\ x).\mathcal{T}).\mathcal{T}_{2,f} \mid (ty_case\ \lambda\overline{x_{(1)}}.\mathcal{T}_1\ \cdots\ \lambda\overline{x_{(k)}}.\mathcal{T}_1\ \mathcal{T})\\
\mathcal{T}_{2,f} &= (\mathcal{T}_1\ f) \mid (f\ \mathcal{T}_{2,f}\ \mathcal{T}_{2,f}) \mid (\mathcal{T}_{2,f}\ \mathcal{T}_{2,f}) \mid \lambda x : \mathcal{T}.\mathcal{T}_{2,f} \mid\\
&\qquad (ty_case\ \lambda\overline{x_{(1)}}.\mathcal{T}_{2,f}\ \cdots\ \lambda\overline{x_{(k)}}.\mathcal{T}_{2,f}\ \mathcal{T}_{2,f})
\end{aligned}$$

The notation $\mathcal{T}_{2,f}$ is used to stress the fact that the functional argument must really be used in $F$ in a more restrictive manner than any other $\lambda$-term. In particular, no instance of $\mathcal{T}_{2,f}$ may be $f$, and $f$ may not occur free in $\mathcal{T}_1$ in the case $(\mathcal{T}_1\ f)$ of $\mathcal{T}_{2,f}$.

## 5 Usage in the Coq system

In the Coq system, the type *Acc* is defined exactly as it is described in this paper. The function $Rec_{wf}$ is represented by a function named *well_founded_definition*. This function takes as parameters the input type (corresponding to $A$ throughout this paper), the relation on this type ($<$), the theorem stating that this relation is well-founded ($wf_<$), a function mapping elements of the input type to sets ($B$), and a function ($F$).

This method to define well-founded recursive function is specially supported in the syntax of the `Program` tactic, that makes it possible to describe an algorithm without giving a complete justification that it is well defined, the necessary proofs being left as side conditions. Because the extraction mechanism provided by Coq uses the sort mechanism to delete proof arguments in functions, well-founded recursive function are naturally extracted as recursive functions that can be used to compute in Caml or a lazy programming language.

Programmers in Coq do not use the *ty_case* operators to write down their algorithms. Instead, the Coq system provides a pattern matching construct that is equivalent to *ty_case* or *ty_case'* constructs, the distinction being done automatically depending on the type of the result. An advantage of this approach is that the same pattern matching construct is used to describe the function and the proof. A drawback is that a single pattern matching construct may expand into a large number of *ty_case* constructs if the patterns occurring on the left hand side of pattern-matching rules are deep.

Also, pattern-matching constructs can be much simpler to write than *ty_case* constructs when the types of all cases are equal. In this case, the user does not need to provide this type, which is computed automatically by the proof system from the type of the first branch. In this case, we will say that the pattern-matching construct becomes simply typed.

We have used our method to construct by hand fixpoint equalities for a small variety of functions including Ackermann's function, well-founded encodings of factorial, remainder, a list-based quicksort, and some auxiliary function working on lists for a Java byte-code verifier. In all the cases treated so far, the functions defined by well-founded recursion had a non-dependent type, so that the problem raised in the second phase (case 2) did not occur.

Based on our method, we are constructing a procedure to generate the equality automatically when it is possible. This procedure will be provided in Coq with the following syntax:

```
Wf_Definition f A B order th_wf F.
```

Where `f` is the name of the new recursive function to be defined, `A` is the input type, `B` is the function that computes the output type from the input value, `order` is a binary relation on `A`, `th_wf` a proof that `order` is well-founded, and `F` the value of the functional that describes the function being defined (the functional of which `f` will be the fix-point, so to say). This command defines the function `f` and the theorem `f_fxp_eqn` whose statement is :

$$\forall x : A.(f\ x) = (F\ x\ \lambda y : A.\lambda h : (order\ y\ x).(f\ y))$$

For instance, here is how our tool is to be used to define the log function.

```
Section define_rem.
Parameters div2 : nat -> nat; th : (n:nat)(lt (div2 n) (S n)).

Wf_Definition log nat [_:nat]nat lt lt_wf
  [x:nat]<[x:nat](f:(y:nat)(lt y x)->nat)nat>Cases x of
       O => [f:?](O)
     | (S n) => [f:?](S (f (div2 (S n))(th n)))
   end.
```

The fixpoint equation we obtain is:

```
log_fxp_eqn :
 (x:nat) (log x)
   =(<[x0:nat]((y:nat)(lt y x0)->nat)->nat>
       Cases x of
        O => [_:((y:nat)(lt y (O))->nat)](O)
       |(S n) =>
        [f:((y:nat)(lt y (S n))->nat)](S (f (div2 (S n))(th n)))
       end [y:nat; _:(lt y x)](log y))
```

But it is easy to obtain a simpler formulation, where dependently typed case expressions become simply typed:

```
Theorem log_simple_equation :
  (x:nat)(log x) =
     (Cases x of O => (O) | (S n) => (S (log (div2 (S n)))) end).
Intros x;Rewrite log_fxp_eqn;Case x;Simpl;Auto.
Qed.
```

## 6 Conclusion

The practical result of this work is a tool to generate an equation that is usually difficult to obtain. In its simplest form, the equation can be as simple as the function definitions that can be obtained with the `Program` tactic, where proof information also disappears [10].

The desire to produce proofs of fix-point equalities without using extensionality looks very much like a theoretic rather than pragmatic question. As mere users of a type-theory based proof system, we do not know how well or how badly the axiom of extensionality interferes with other aspects of the logic we use. The theoretical result brought by this paper is that we do not need to answer this question for a reasonable class of functions.

In future work, we would like to study how this method can be extended to handle functions containing recursion operators and mutual recursion. We also want to provide a simplified version of the fix-point equation where proof arguments do not appear, following the example of `log_simple_equation`. This improvement can be implemented by a simple partial evaluation. It should also be possible to re-use the results of Parent [10] and Slind [15] towards the support of plain functional programming in proof systems.

## References

1. Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, 1977.
2. Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.
3. Robert L. Constable and Nax P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *LNCS*, pages 61–78. Springer Verlag, June 1985.
4. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
5. Thierry Coquand and Henrik Persson. Gröbner bases in type theory. In *Types'98*, volume 1658 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
6. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

7. John Harrison. Inductive definitions: Automation and application. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Provoing and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 1995.
8. Thomas F. Melham. A package for inductive relation definitions in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving system and its Applications*, pages 350–357. IEEE Computer Society Press, 1992.
9. Bengt Nordström. Terminating general recursion. *BIT*, 28, 1988.
10. Catherine Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1995.
11. Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.
12. Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. Technical report 57, University of Cambridge, Computer Laboratory, October 1984.
13. Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
14. Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. technical report CMU-CS-89-209.
15. Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*. Sprinter Verlag, August 1996.
16. Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1998.

## A Proof of the transfer theorem

This text can be checked mechanically using Coq Version 6.3.1.

```
Scheme Acc_ind2 := Induction for Acc Sort Prop.
Transparent well_founded_induction.
Variables A:Set; R:A -> A -> Prop;
   R_wf:(well_founded A R); B:A -> Set;
   F:(x : A) ((y : A) (R y x) -> (B y)) -> (B x).
Definition f := (well_founded_induction A R R_wf B F).

Inductive Trans  : A -> A -> Prop :=
  Tc1: (x, y : A) (R x y) -> (Trans x y)
 | Tc2: (x, y, z : A) (R x y) -> (Trans y z) -> (Trans x z).
Hints Resolve Tc1 Tc2 ex_intro.

Theorem trans_balance:
```

```
  (x, y, z : A) (R x y) -> (Trans y z)
    -> (Ex [t : A] (Trans x t) /\ (R t z)).
Intros x y z H H0; Generalize x H; Clear H x; Elim H0;EAuto.
Intros x y0 z0 H H1 H2 x0 H3; Elim (H2 x); Auto.
Intros x1 H4; Exists x1;Elim H4;EAuto. Qed.

Theorem well_founded_ind2:
 (P : A -> Prop)
  ((x : A)((y : A)(Trans y x) -> (P y)) -> (P x)) -> (a : A)(P a).
Intros P H a; Apply H; Cut ((y : A)(Trans y a) -> (P y)) /\ (P a).
Intros H0; Elim H0; Auto.
Elim a using (well_founded_ind A R R_wf).
Intros x H0; Split. Intros y H1; Inversion H1. Elim (H0 y); Auto.
Elim trans_balance with 1 := H2 2 := H3.
Intros x1 H7; Elim H7; Intros H8 H9. Elim (H0 x1); Auto.
Apply H; Intros y H1; Generalize H0; Inversion H1.
Intros H5; Elim (H5 y); Auto.
Elim trans_balance with 1 := H2 2 := H3.
Intros x1 H6 H7; Elim H6; Intros H8 H9.
Elim H7 with y := x1; Auto.  Qed.

Theorem is_trans:
 (x, y, z : A) (Trans x y) -> (R y z) -> (Trans x z).
Induction 1; EAuto.
Qed.
Hints Resolve is_trans.

Theorem wf_transfer:
 ((x : A) (f' : (y : A)  (B y))
          (g : (y : A) (R y x) -> (B y))
          ((y : A) (h : (R y x))  (g y h) = (f' y))
   -> (F x [y : A] [h : (R y x)]  (g y h)) =
      (F x [y : A] [_ : (R y x)]  (f' y)))
 -> (x : A)  (f x) = (F x [y : A] [h : (R y x)] (f y)).
Unfold f; Intros H x; Elim x using well_founded_ind2; Intros x'.
Unfold 3 well_founded_induction.
Elim (R_wf x') using Acc_ind2; Simpl.
Intros x'' accessible H_rec_acc H_rec_well_founded.
Apply H with
  g := [y : A] [h : (R y x'')]
       (Acc_rec A R B [x0 : A]
          [_ : (y1 : A) (R y1 x0) -> (Acc A R y1)]
          [g' : (y1 : A) (R y1 x0) -> (B y1)]
          (F x0 g') y (accessible y h)).
Intros y h; Rewrite H_rec_well_founded; EAuto. Qed.
```